Managing Hardware Verification Complexity with Aspect-Oriented Model-Driven Engineering

Éamonn Linehan, Siobhán Clarke

***Abstract*—Recent advances in both the capabilities and accessibility of embedded systems have resulted in the potential to build increasingly complex systems that consequently are difficult to develop, test and deploy. Model-driven approaches raise the level of abstraction at which developers work, promising improved quality (reliability, safety, real-time properties) and increased productivity through automation. However, despite the increasing application of model-driven technologies to the development of embedded systems, little attention has been paid to the corresponding increase in complexity of verification environments for embedded systems. As system complexity has increased in recent years so has the complexity of hardware verification testbenches resulting in them becoming difficult to understand, maintain, extend and reuse across projects. This paper presents a new UML profile for the e verification language that enables the use of an aspect-oriented, model-driven approach for the design of verification testbenches.**

***Index Terms*—simplicity, beauty, elegance**

I. Introduction

EMBEDDED systems have become increasing complex in recent years as improvements in hardware performance and reductions in cost have enabled construction of more technically advanced systems. In addition, embedded systems require a higher level of robustness and reliability because they control real-world physical processes or devices upon which we depend, frequently in a critical way. Consequently, methods for developing and modeling embedded systems and rigorously verifying behavior are increasingly important.

In the case of embedded systems that are realised in hardware, the verification process has become a time consuming design process that is estimated to occupy up to 70% of design time with hardware verification testbenches making up 80% of the total code generated during a project [4]. As embedded systems become more complex and time-to-market requirements become shorter, extra pressure is placed on verification engineers to complete exponentially more complex verification projects in shorter time periods.

In hopes of increasing productivity, verification engineers seek to work at higher levels of abstraction and increase their ability to reuse testbenches. In response to this demand hardware verification languages have evolved from low level C libraries to aspect and object oriented domain specific languages with built in support for functional coverage measurement, constrained random generation and other verification specific functionality. However, these languages present verification engineers with new challenges as tool support remains limited, methodologies and development process do not take full advantage of the power of these languages and the use of programming paradigms unfamiliar to hardware engineers continues to limit their ability to understand, maintain, extend and reuse code across projects.

Model-driven engineering is a design approach where systems are specified as models. Depending on the level of abstraction of the model, code can be generated ranging from system skeletons to complete, deployable products. Model-driven engineering aims to increase productivity by simplifying the design process and promoting communication between engineers working on the system. However, there has been limited application of model-driven engineering approaches to the challenges of hardware verification [21]. Where model-driven approaches have been taken the models themselves are often only used as documentation and are incapable of representing all the features available in the target verification language. Where code generation is possible, it is generally limited to structural skeletons and cannot interact with legacy components.

This paper presents a meta-model for the *e* hardware verification language that can be used as part of a model-driven engineering toolset for embedded systems development. The meta-model is implemented as an extension to UML, incorporating aspect-oriented constructs from Theme/UML [6] and design and verification constructs from MARTE [19] to help engineers organise code in a way that makes it easy to deal with the concerns they really care about in a verification environment. For example, it is no longer necessary to organise all code into objects. Theme/UML constructs allow engineers to organise code around functionality, layers, protocols, coverage or any other verification concern that is important to them [20]. The toolset supports a development process for the separation of embedded systems concerns in verification and subsequent code generation, accelerating the automated development of reusable verification environments.

The rest of the paper is structured as follows. Section II introduces the challenges that distinguish hardware verification from traditional software engineering and introduces the *e* hardware verification language. Sections III and IV describe the relevant concepts in the Theme/UML aspect-oriented design language and its application as part of a model-driven engineering toolset to reduce complexity in embedded systems verification through aspect-oriented modularisation. Section V presents the UML profile for the *e* verification language illustrating how it can be used to model hardware verification testbenches. Section VI summarises related work while Section VII concludes the paper and outlines plans for

Lero - The Irish Software Engineering Research Centre, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Ireland.

future work.

II. Hardware Verification and e programming language

Verification is the process of demonstrating that the implementation of a design matches its specification. Figure 1 illustrates a typical design process in which a system is initially specified as textual descriptions. From this design description an engineer produces an implementation in a hardware description language. In parallel, testbenches are written to verify that the functionality of the implementation matches the design. The term "testbench" refers to simulation code used to create a predetermined input sequence to a design, then optionally to observe the response. A testbench is commonly implemented using VHDL, Verilog, *e* or OpenVera, but it may also include external data files or C routines [4]. Once all functional tests are passed, the implementation can be synthesised to a circuit design that can be manufactured. This process can involve many engineers with limited cross-over between teams writing verification testbenches and teams implementing the design.

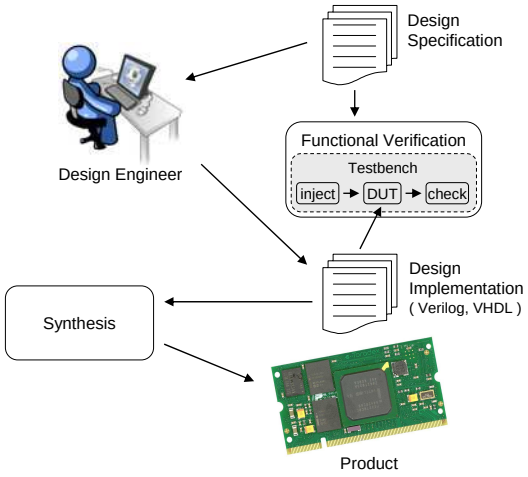

Figure 1. Functional Verification Process

In recent years, significant advances in chip and system fabrication technologies have afforded designers the ability to implement digital systems with ever increasing complexity. A consequence of this is that the process of verifying these new systems has also increased in complexity. In 2003 Bergeron reported that 70% of system design effort goes to verification [4], while in 2007, Li et al. asserted that up to 80% of design costs in many circuit design projects are due to verification [14]. Verification has become a time consuming process that limits the speed at which new products can be developed. This is a problem know as the *design productivity gap*, that is, the difference between the number of transistors that can be manufactured on a chip with the number of transistors engineers can take advantage of in a reasonable amount of time.

The *productivity gap* can only be addressed by increasing productivity through automation, reuse, and by moving to higher levels of abstraction. An example of a move to higher levels of abstraction is the move from manually created, project specific test suites that did not scale to verification-specific programming languages. These languages have verification-specific constructs as primitives and built-in capabilities to perform pseudo-random test generation.

The *e* hardware verification language is one such domain-specific programming language that was developed in 1997 by Verisity Design (subsequently acquired by Cadence Design Systems [1]) as part of their Specman tool [13]. *e* was standardised as IEEE 1647 [12] and a second revision of the standard was published in 2008. In this section we introduce the *e* language by describing its key concepts and examining its aspect-oriented features. At *e*'s core is a pseudo-random generator that facilitates creation of input stimuli that can be applied to the design under test (DUT). All variables are assigned a random value unless either marked as not generatable or constrained to be a specific value. *e* contains constructs that allow the response of the DUT to be monitored and checked. In addition, there are constructs to support assessment of the functional coverage of the DUT (as opposed to simply the code coverage).

In providing support for the development of testbenches, *e* brings together concepts from several languages [24]:

- It has a basic object-oriented (OO) programming model with automatic memory management and single inheritance in a similar manner to Java.
- *e* natively supports aspects.
- *e* supports constraints as object features, using constraints to refine object models. The execution model resolves the constraints, picking random values that satisfy the constraint set.
- *e* is strongly typed, like Pascal and Modula.
- *e* has concurrency constructs for hierarchical composition, similar to hardware description languages such as Verilog and VHDL.
- *e* contains temporal logic constructs that borrow from linear temporal logic and interval temporal logic.

The *e* languages native support for aspects is one of its strengths. Aspect-orientation is an alternative way to modularise software that is usually used in conjunction with object orientation. An aspect is the name given to a grouping of related functions and variables that are related to a single concern. A concern is anything a software developer may in interested in and wish to encapsulate so that it can later be reused, reviewed or replaced. Separation of concerns is an established technique for managing system complexity though modularisation of distinct concerns [8]. Some concerns are crosscutting, that is, they exists in multiple places in the code and cannot be entirely separated from other concerns in a purely object-oriented decomposition. Aspect-oriented languages extend the modularisation capabilities of the object-oriented paradigm and have demonstrated effectiveness at separating crosscutting concerns in a wide range of domains [11], [17], [25].

Aspect-oriented programming is a powerful mechanism

when applied to verification [4]. The AOP features of the *e* language give it the power to significantly simplify and accelerate the development of reusable, automated, verification environments [20]. However, aspect-orientation in *e* is rarely considered at design time as a way of modularising code. Instead, aspect-oriented features of *e* are often used to cleanly add new features to existing code without having to intrusively modify the code base. This can in part be attributed to the *e* Reuse Methodology [13] that advocates an OO decomposition at design time but also the limitations of *e*'s aspect-oriented features and challenges in managing a global software development process. In previous collaborations with industry we have found that testbenches are often developed by hardware engineers that have no formal training in software development, resulting in insufficient documentation and difficulties in reusing components, particularly where the original author is not available to consult [9]. These challenges are further exacerbated by organisational demands to reduce the time taken to perform verification as it does not contribute directly to the end product and consumes such a large portion of the products development time.

III. Theme/UML

The Theme Approach is an aspect-oriented methodology that encompasses the requirements analysis, design and mapping to implementation phases of the development lifecycle [6]. Theme/UML is an aspect-oriented modeling language facilitates graphical modeling of concerns in an extended version of UML.

Theme/UML augments standard UML with new modularisation and compositional constructs. Figure 2 shows how base themes and aspect themes are designed. Base themes are modeled using the standard UML process and diagram types. An aspect theme is one that encapsulates a crosscutting concern and is designed relative to abstract templates. UML behavioural diagrams are used to specify when and how the templates interact with the base themes.

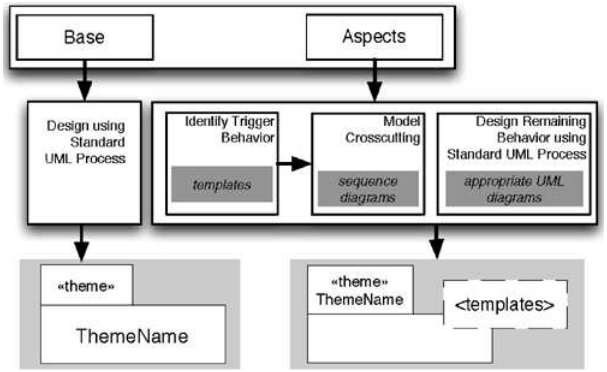

Figure 2. Designing with Theme/UML

IV. Model-Driven Engineering

Model-driven engineering is an approach to software development that focuses on the production of high-level models that are used as the basis for automating system implementation. Outside the embedded systems world, model-based design has become a popular object-oriented software development paradigm, primarily due to the publication of the Object Management Group's Model-Driven Architecture (MDA) [16]. Although MDA originally targeted large-scale enterprise applications, its underlying principles have been widely adopted and applied to the development of applications in a diverse range of domains, including hardware design and verification. The MDA approach centers on the definition of a Platform Independent Models (PIMs) using a high-level specification language. The goal is to develop models that are precise enough to support code generation so that a PIM may be transformed into one or more Platform Specific Models (PSMs) for the actual implementation. The advantage of the MDA approach is that models and code are more easily kept up to date, incremental, iterative development is facilitated by the direct transformation from model to code and increased automation in the production of system code reduces the potential for the introduction of human errors.

Model-Driven Theme/UML is a set of model-driven tools with a supporting development process that facilitates modularised design with Theme/UML and subsequent model composition and synthesis to source code [5]. In this work we extend the model-driven Theme/UML toolset by introducing a UML profile for *e*.

V. Modeling the *e* Language in Theme/UML

In collaboration with Infineon Technologies, we have previously analysed the challenges in modeling hardware verification environments, using Theme/UML [9]. Theme/UML is a natural fit to modeling *e* testbenches because of its aspect-oriented approach. However, when applied to verification, the extended Model-Driven Theme/UML approach has been found to be deficient in a number of areas. Specifically, temporal concerns, runtime constrained composition, constraints and type extension could not be easily modeled.

To capture these features in UML, a new profile[1] for the *e* verification language has been defined. The *e* UML Profile is a collection of such extensions that collectively customize UML for the hardware verification domain, in particular hardware verification with the *e* verification language.

The *e* UML profile inherits features from both Theme/UML and the OMG UML profile for Modeling and Analysis of Real-time and Embedded systems (MARTE). Figure 3 illustrates the relationships between these profiles. Elements from the Time, Value Specification Language components of MARTE are extended and reused in addition to the full set of features of the Theme/UML profile. The *e* profile itself is divided into three packages: the Core package contains model elements corresponding to *e* language constructs; the Verilog and VHDL packages contain simulation related constructs (statements or unit members that expose some functionality of the Verilog / VHDL simulator interface).

Table I lists the mapping from *e* language constructs to their UML representation. The grouping of language constructs is

[1] A UML profile is an extension mechanism for customizing UML models for particular domains and platforms. Profiles are defined using stereotypes, tag definitions, and constraints that are applied to specific model elements.

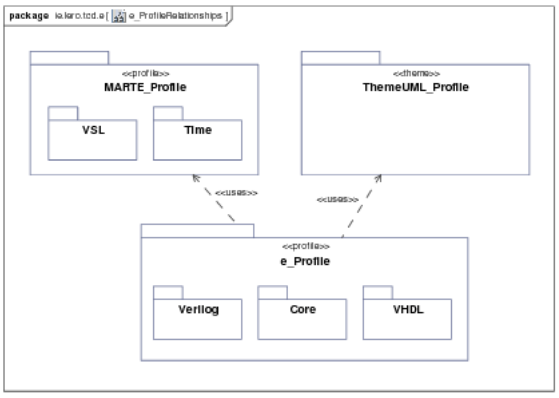

Figure 3. UML profile hierarchy

taken from IEEE 1647 [12] with UML elements denoted by `UML::` referring to UML2 meta classes. The UML profile named "`e_Profile`" is contained within the namespace `ie.lero.tcd.e` (Not Shown). In addition to the elements in the table above, a complete set of constants, predefined elements and the standard struct hierarchy are contained within the profile.

The ability to model the *e* language using the modeling constructs defined in Table I is illustrated in the following two examples. The first example illustrates basic structural elements of the *e* language including an enumerated scalar type and struct containing fields.

```
type NetworkType: [IP=0x0800, ARP=0x8060]
(bits: 16);
struct header {
  hdr_type: NetworkType;
  %len: int;
};
```

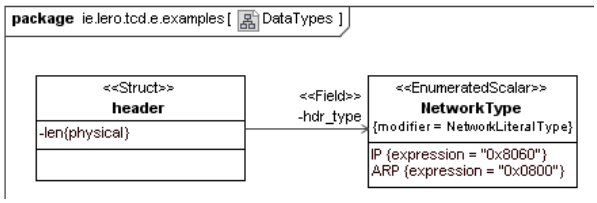

Figure 4. Example of *type* and *struct* modeling

Figure 4 illustrates the UML class diagram that corresponds to the source listing above. This diagram was drawn in the software modeling tool MagicDraw 16.5 Academic Version by importing the `e_Profile` module into a new project. The second example illustrates the use of Theme/UML to model the aspect-oriented extension features of *e*. This example assumes a struct has previously been defined with the name *packet* and is extended in a new module *packet_extsn*.

```
packet{
  show() is only {
    out("This packet has an error...");
  };
}
```

e Construct	UML Extension
Module	Based on style recommendation by Robinson [20] who recommends that files are organised by concern. Each module is modeled as an extension to `ThemeUML_Profile::theme`. Module imports are modeled as extensions to `UML::Dependency`.
Scalar Subtypes	Primitive types are specified as `Primitives` in package `e_Profile::e_PrimitiveTypes`. Subtypes are specified in *e* by using a scalar modifier to specify the range or bit width of a scalar type. These are modeled as extensions to metaclass `UML::PrimitiveType` in package `e_Profile::e_ScalarTypes`.
Enumerated Types	Specified as `EnumeratedScalar` extending metaclass `UML::Enumeration` with a `e_Profile::WidthModifier` tag. Lists are modeled as extensions of `MARTE_Profile::-VSL::DataTypes::collectionType`.
Struct / Unit	Specified as extensions to metaclass `UML::Class` in package `e_Profile::Core`
Fields	Fields are specified as extensions to `UML::NamedElement` that inherit from `e_-Profile::Core::Struct::StructMember`. Attributes are modeled as properties of `e_Profile::Core::Field`.
Like Inheritance	Single inheritance specified as extension to `UML::Generalisation`
When Inheritance	Specified as an extension to `UML::Generalisation` with tags to hold reference to the base struct type field.
Type Extension	Implemented as an extension to `ThemeUML_Profile::theme` composition semantics to facilitate the merge of enumerated types.
Time Consuming Methods	Methods and inline methods are modeled as extensions of `UML::Operation`. TCM's also inherit from `MARTE_Profile::Time::TimedElement` and contains a property of type `e_Profile::Core::Event::HDLEvent` (representing a Verilog clock expression).
AO Extension	*is also,first, only* and *inline only* extensions are modeled as templated operations within a `ThemeUML_Profile::theme`. The theme profile is extended to support an additional composition relation to model type extension which can be constraining rather than additative.
Constraints	Specified as an extension to `UML2::Classes::Kernel::Constraint`
Coverage	A coverage group is modeled as a `e_Profile::Core::StructMember` containing a collection of `CoverItem`'s and a reference to an `e_Profile::Core::e_Event`.
Ports	Modeled as extended `MARTE_Profile::GCM::Interaction_Port` including tagged values for `port-kind`.

Table I
MAPPING *e* VERIFICATION LANGUAGE TO UML EXTENSIONS.

Figure 5 illustrates how Theme/UML can be used to graphically represent e's is also extension. The aspect-oriented extension, called *packet_extsn*, is designed so that the cross-cutting behaviour contained in the method *showMore()* is executed after any method from the base system (represented in this example by *show()* in the base theme bound to the aspect). This example shows how Theme/UML can be used to graphically represent e's *is also* extension. The *is first* and *is only* extensions can be modeled in a similar fashion.

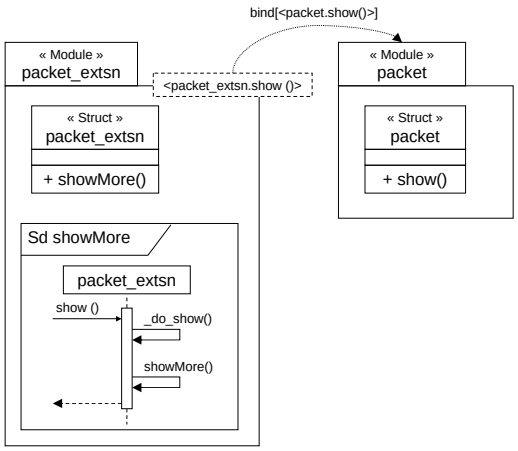

Figure 5. An aspect-oriented method extension in *e*

VI. Related Work

There are a number of approaches that provide some support for model driven engineering of hardware verification testbenches. In this section we review a number of related approaches that model hardware definition or verification languages.

There area set of approaches that are capable of generating UML or other models from verification testbenches for the purpose of documenting their design. For example, the design and verification tools plugin (DVT) for eclipse by AMIQ [3] is capable of extracting a UML class diagram illustrating a set of selected structs from an *e* or SystemVerilog testbench. The purpose of this feature is to extract a documenting UML model that shows inheritance, associations (pointers) and class members. The ability to diagrammatically illustrate the structure of the testbench using a tool for drawing directed graphs reduces the time it takes engineers to become familiar with a new code base and helps manage naming and code navigation. However, the UML diagrams produced do not contain any behavioural information and cannot capture the aspect-oriented constructs of the *e* language, making the diagrams of little use for interpreting behaviour at runtime. The transformation from code to model cannot be reversed preventing any modifications of the class diagrams being reflected in the code.

Other work by Thompson et al. goes further than simply facilitating the generation of documentation and provides some support for code generation [23]. However in this case code stubs for the verification language Vera are generated from class diagrams using UML to C++ code synthesis. The code skeletons then need to be modified by hand to remove C++ specific artefact's and have their behaviour inserted. Because there is no Vera specific model the transformations are not fully automated and cannot be reversed.

The UML to SystemVerilog synthesis proposed by Li et al. takes a different approach [14]. UML is extended with a profile supporting the modeling of real-time systems and the ability to informally specify verification assertions at the model level is added. UML state diagrams are then transformed using an intermediate XMI representation to SystemVerilog code. Both structure and behaviour are specified at the model level but the transformation is not reversible. McUmber at al. also make use of UML class and state diagrams to specify both structure and behaviour [15]. In their case VHDL specifications are generated by applying a set of rules for mapping from UML to VHDL.

These early examples of modeling hardware definition and verification languages had the ambition of reducing design complexity by raising the level of abstraction engineers work at, enabling designs so complex that they cannot be understood in detail all at once to be broken down into pieces that can be viewed and understood, one aspect at a time. However, these approaches fail to hide implementaiton details. Model driven approaches however increase the level of automation and facilitate the specification and transformation of models at multiple levels of detail as well as reversible model to code transformations. The work by Coyle et al. is an early example of the application of model-driven engineering to the area of hardware design and verification [7]. The Test-Bencher Pro graphical code generator by SynaptiCAD Inc., inspired by model-driven engineering's platform independent models, provides a means to model verification testbenches independent of the verification language in use [22]. However, TestBencher Pro's timing diagram can only model a subset of the functionality required of a verification testbench.

Further reductions in design complexity can be achieved by separating models into smaller more coherent pieces. Aspect-oriented modeling provides a mechanism to achieve this by introducing a new way to modularise models that is used in addition to traditional object-oriented modeling. Although there has been some work in aspect-orientated and model-driven engineering of embedded systems [2], [10], these approaches can not be directly applied to hardware design and verification as languages in these areas incorporate constructs that do not appear in general purpose high level languages (like C++ or Java). For example, constrained random stimulus generation, temporal assertions and functional coverage constructs.

VII. Conclusion & Future Work

Model-driven approaches raise the level of abstraction at which developers work, promising improved quality (reliability, safety, real-time properties) and increased productivity through automation. However, despite the increasing application of model-driven technologies to the development of embedded systems, little attention has been paid to the corresponding increase in complexity of verification environments. This paper has presented a new UML profile for the *e* verification language. This profile, when used as part of a model-driven engineering process, can enable the design and development of verification environments at the model level, reducing the cost of verifying hardware designs and ultimately reducing the time to market for new products. The aspect-orientated nature of the modeling language facilitates greater reuse by making it easier to exploit the aspect-oriented constructs built into the *e* verification language.

A limitation of the current *e* profile is how close it is to the code level, requiring engineers to first have a good knowledge of the *e* verification language. The extraction of verification features and constructs that are common to all verification languages into a higher level platform independent model will facilitate the design of verification testbenches at a higher level of abstraction and will eliminate the implementation and verification language specific features that are present in the current *e* UML profile. This future work will further contribute to the approaches ability to reduce design complexity.

In addition to this new model, we plan to extend to the model-driven process we have defined for the construction of embedded system software to include verification. This is achieved through the use of SysML [18] requirements diagrams which relate aspect-oriented system designs and testbenches to source requirements. For example, Theme/UML system design models can be related to requirements through SysML derive relationships that indicate where a theme is derived from a particular requirement. Similarly, verify relationships determine which testbench modules (also modeled as themes) fulfill a particular requirement.

VIII. Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

References

[1] Cadence design systems. Online; accessed 8 February 2010. http://www.cadence.com/.

[2] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–8, New York, NY, USA, 2008. ACM.

[3] AMIQ Consulting. DVT - the complete development environment for e and systemverilog, February 2010. Online; accessed 9 February 2010; http://www.dvteclipse.com/.

[4] Janick Bergeron. *Writing Testbenches - Functional Verification of HDL Models*. Springer - Verlag, 2 edition, 2003.

[5] Andrew Carton, Cormac Driver, Andrew Jackson, and Siobhán Clarke. Model-driven theme/uml. *Transactions on Aspect-Oriented Software Development VI: Special Issue on Aspects and Model-Driven Engineering*, pages 238–266, 2009.

[6] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, NJ, 1st edition, March 2005.

[7] Frank P. Coyle and Mitchell A. Thornton. From uml to hdl: a model driven architectural approach to hardware-software co-design. In *Proceedings of Information Systems: New Generations Conference (ISNG)*, pages 88–93, April 2005.

[8] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[9] Darren Galpin, Cormac Driver, and Siobhán Clarke. Modelling hardware verification concerns specified in the e language: an experience report. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 207–212, New York, NY, USA, 2009. ACM.

[10] Sébastien Gérard, Jean-Philippe Babau, and Joël Champeau, editors. *Model Driven Engineering for Distributed Real-time Embedded Systems*. Hermes Science Publishing, 2005.

[11] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *ECOOP 2007 - Object-Oriented Programming*, volume 4609/2007. Springer, 2007.

[12] IEEE Computer Society. Ieee std 1647-2008, ieee standard for the functional verification language e. Standard IEEE Std 1647-2008, IEEE, NY, USA, August 2008.

[13] Sasan Iman and Sunita Joshi. *The e Hardware Verification Language*. Kluwer Academic, Norwell, MA, USA, 2004.

[14] Lun Li, Frank P Coyle, and Mitchell A Thornton. uml to systemverilog synthesis for embedded system models with support for assertion generation. In *Proceedings of the ECSI Forum on Design Languages*, September 2007.

[15] William E. McUmber and Betty H. C. Cheng. Uml-based analysis of embedded systems using a mapping to vhdl. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 56–63, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[17] Jennifer Munnelly, Serena Fritsch, and Siobhan Clarke. An Aspect-Oriented Approach to the Modularisation of Context. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, pages 114–124, Washington, DC, USA, 2007. IEEE Computer Society.

[18] OMG. SysML - systems modeling language. v1.1, January 2008. formal/2008-11-01. /urlhttp://www.omg.org/spec/SysML/1.1/.

[19] OMG. A UML profile for MARTE, June 2008. ptc/08-06-09. /urlhttp://www.omgmarte.org/Documents/Specifications/08-06-09.pdf.

[20] David Robinson. *Aspect-oriented Programming with the e Verification Language - A Pragmatic Guide for Testbench Developers*. Morgan Kaufmann, MA, USA, 2008.

[21] Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. *Computer*, 42:53–59, 2009.

[22] SynaptiCAD. Testbencher pro, 2010. URL: http://www.syncad.com/testbencher_verilog_vhdl_testbench_generator.htm.

[23] Kevin Thompson and Ladd Williamson. Hardware verification with the unified modeling language and vera. In *Proceedings of the Synopsys User Group San Jose*, 2002.

[24] Verisity Design, Inc. e language reference manual, February 2002. Online; Accessed 8th February 2010; http://www.ieee1647.org/downloads/prelim_e_lrm.pdf.

[25] Marco A. Wehrmeister, Edison P. Freitas, Carlos E. Pereira, and Flavio R. Wagner. An Aspect-Oriented Approach for Dealing with Non-Functional Requirements in a Model-Driven Development of Distributed Embedded Real-Time Systems. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 428–432, Washington, DC, USA, 2007. IEEE Computer Society.